# Unit – I

**Syllabus:**

- Data Structures - Definition, Classification of Data Structures, Operations on Data Structures, Abstract Data Type (ADT), Preliminaries of algorithms. Time and Space complexity.
- Searching - Linear search, Binary search, Fibonacci search.
- Sorting- Insertion sort, Selection sort, Exchange (Bubble sort, quick sort), distribution (radix sort), merging (Merge sort) algorithms.

## INTRODUCTION:

- A *data structure* is a particular way of storing and organizing data in a computer so that it can be used efficiently.
- Some common examples of data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables
- Today computer programmers do not write programs just to solve a problem but to write an efficient program.
- When selecting a data structure to solve a problem, the following steps must be performed.
  1. Analysis of the problem to determine the basic operations that must be supported.
  2. Quantify the resource constraints for each operation.
  3. Select the data structure that best meets these requirements.

- The term *data* means a value or set of values. It specifies either the value of a variable or a constant (e.g., marks of students, name of an employee, address of a customer, value of *pi*, etc.).
- A *record* is a collection of data items. For example, the name, address, course, and marks obtained are individual data items. But all these data items can be grouped together to form a record.

A *file* is a collection of related records. For example, if there are 60 students in a class, then there are 60 records of the students. All these related records are stored in a file.

To develop a program of an algorithm we should select an appropriate data structure for that algorithm. Therefore, data structure is represented as:

**Algorithm + Data structure = Program**

## CLASSIFICATION OF DATA STRUCTURES:

- Data structures are generally categorized into two classes: *primitive* and *non-primitive* data structures.

***Primitive and Non-primitive Data Structures:***

- Primitive data structures are the fundamental data types which are supported by a programming language. Some basic data types are integer, real, character, and boolean. The terms ‘data type,

basic data type', and 'primitive data type' are often used interchangeably.Non-primitive data structures are those data structures which are created using primitive data structures. Examples of such data structures include linked lists, stacks, trees, and graphs.

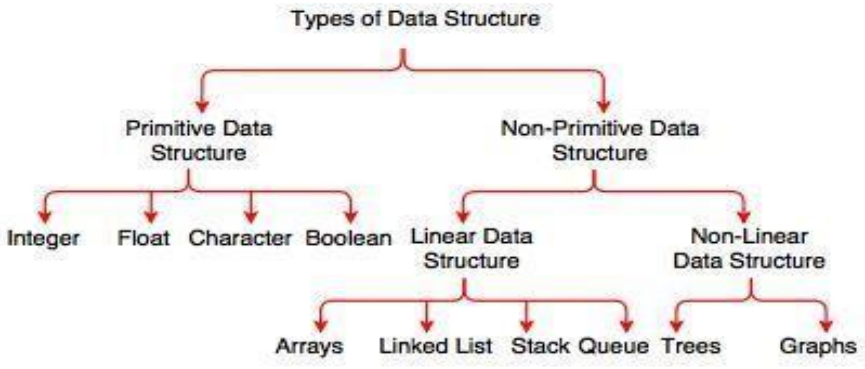



Fig. Types of Data Structure

Non-primitive data structures can further be classified into two categories: *linear* and *non-linear* data structures.

***Linear and Non-linear Structures:***

- If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure.
    - Examples include arrays, linked lists, stacks, and queues.
    - Linear data structures can be represented in memory in two different ways. One way is to have to a linear relationship between elements by means of sequential memory locations. The other way is to have a linear relationship between elements by means of links.

- If the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure.
    - The relationship of adjacency is not maintained between elements of a non-linear data structure. Examples include trees and graphs.

**Arrays:**

- An array is a collection of similar data elements. These data elements have the same data type. The elements of the array are stored in consecutive memory locations and are referenced by an *index* (also known as the *subscript*).
- In C, arrays are declared using the following syntax: datatype name[size];
  Ex: int marks[10];

| 1st element | 2nd element | 3rd element | 4th element | 5th element | 6th element | 7th element | 8th element | 9th element | 10th element |
|---|---|---|---|---|---|---|---|---|---|
| marks[0] | marks[1] | marks[2] | marks[3] | marks[4] | marks[5] | marks[6] | marks[7] | marks[8] | marks[9] |

**limitations**:

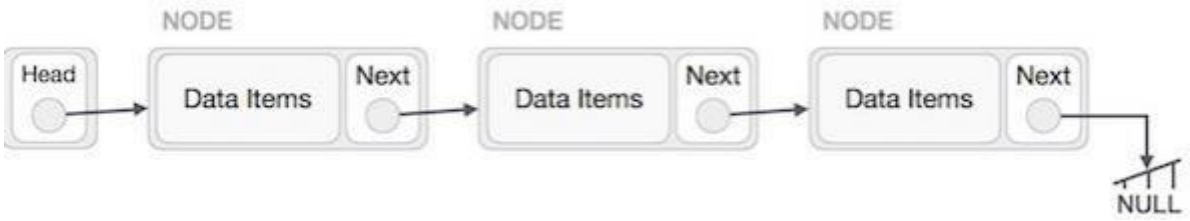
- Arrays are of fixed size.
- Data elements are stored in contiguous memory locations which may not be always available.
- Insertion and deletion of elements can be problematic because of shifting of elements from their positions.

**Linked Lists:**

- linked list is a dynamic data structure in which elements (called *nodes*) form a sequential list.
- In a linked list, each node is allocated space as it is added to the list. Every node in the list points to the next node in the list.
- Every node contains the following

  The value of the node or any other data that corresponds to that node

  A pointer or link to the next node in the list



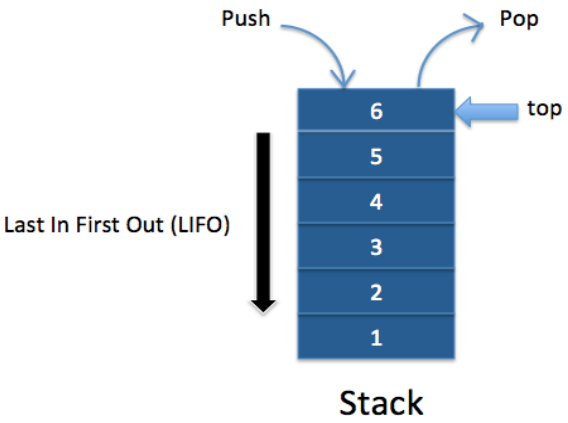

- The first node in the list is pointed by Head/Start/First. The last node in the list contains a NULL pointer to indicate that it is the end or *tail* of the list.

  *Advantage*: Easier to insert or delete data elements

  *Disadvantage*: Slow search operation and requires more memory space

**Stacks:**

- A stack is a linear data structure in which insertion and deletion of elements are done at only one end, which is known as the top of the stack.
- Stack is called a last-in, first-out (LIFO) structure because the last element which is added to the stack is the first element which is deleted from the stack.
- Stacks can be implemented using arrays or linked lists.
- Every stack has a variable top associated with it. Top is used to store the address of the topmost element of the stack.



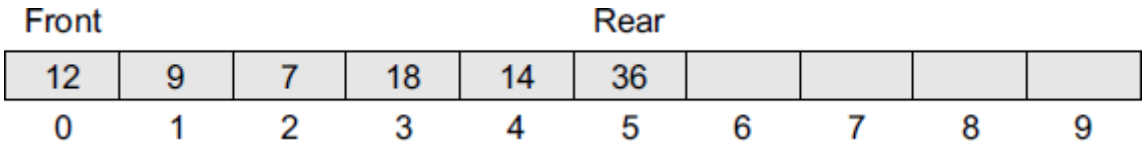

- It is this position from where the element will be added or deleted. There is another variable MAX, which is used to store the maximum number of elements that the stack can store.
- If top = NULL, then it indicates that the stack is empty and if top = MAX–1, then the stack is full.
- A stack supports three basic operations: push, pop, and peep. The push operation adds an element to the top of the stack. The pop operation removes the element from the top of the stack. And the

peep operation returns the value of the topmost element of the stack (without deleting it).

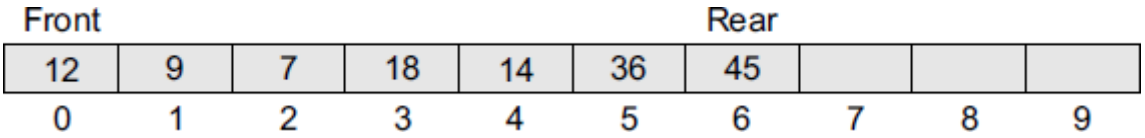
**Queues:**

- A Queue is a linear data structure in which insertion can be done at rear end and deletion of elements can be dome at front end.
- A queue is a first-in, first-out (FIFO) data structure in which the element that is inserted first is the first one to be taken out.

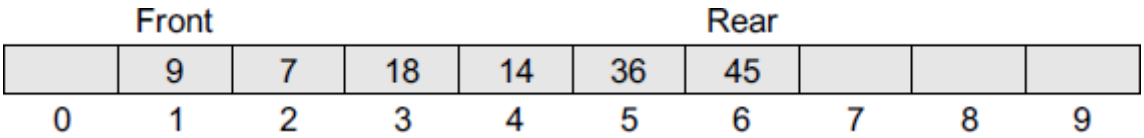

- Like stacks, queues can be implemented by using either arrays or linked lists.




Insert element into the Queue:



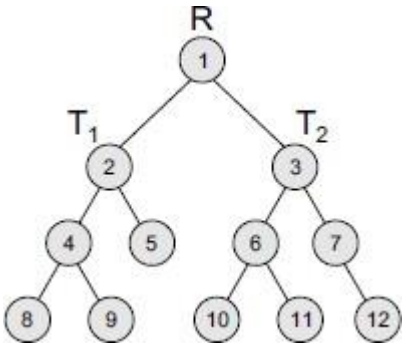

Delete element from Queue:




- A queue is full when rear = MAX – 1, An underflow condition occurs when we try to delete an element from a queue that is already empty. If front = NULL and rear = NULL, then there is no element in the queue.

**Trees:**

- A tree is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical order.
- One of the nodes is designated as the root node, and the remaining nodes can be partitioned into disjoint sets such that each set is a sub-tree of the root
- The simplest form of a tree is a binary tree. A binary tree consists of a root node and left and right sub-trees, where both sub-trees are also binary trees.
- Each node contains a data element, a left pointer which points to the left sub-tree, and a right pointer which points to the right sub-tree.
- The root element is the topmost node which is pointed by a ‘root’ pointer. If root = NULL then the tree is empty.



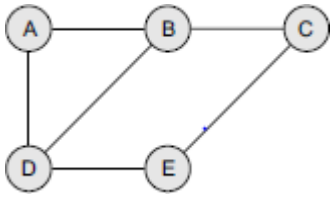
- Here R is the root node and T1 and T2 are the left and right subtrees of R. If T1 is non-empty, then T1 is said to be the left successor of R. Likewise, if T2 is non-empty, then it is called the right successor of R.

  *Advantage*: Provides quick search, insert, and delete operations

  *Disadvantage*: Complicated deletion algorithm

**Graphs:**

- A graph is a non-linear data structure which is a collection of *vertices* (also called *nodes*) and *edges* that connect these vertices.
- A node in the graph may represent a city and the edges connecting the nodes can represent roads.
- A graph can also be used to represent a computer network where the nodes are workstations and the edges are the network connections.




- Graphs do not have any root node. Rather, every node in the graph can be connected with every another node in the graph.

  *Advantage*: Best models real-world situations

  *Disadvantage*: Some algorithms are slow and very complex

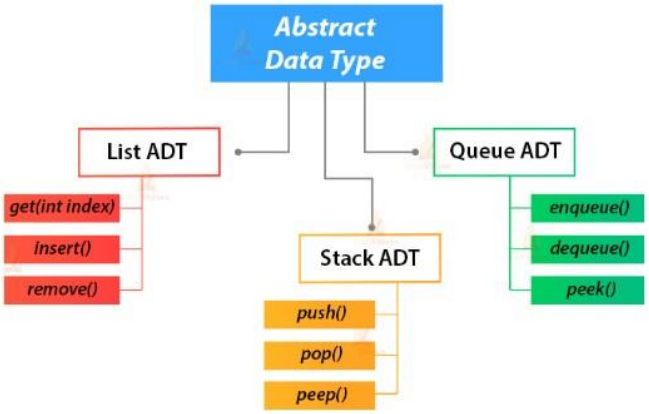
**OPERATIONS ON DATA STRUCTURES:**

- This section discusses the different operations that can be performed on the various data structures previously mentioned.
- ***Traversing*** It means to access each data item exactly once so that it can be processed. For example, to print the names of all the students in a class.
- ***Searching*** It is used to find the location of one or more data items that satisfy the given constraint. Such a data item may or may not be present in the given collection of data items. For example, to find the names of all the students who secured 100 marks in mathematics.
- ***Inserting*** It is used to add new data items to the given list of data items. For example, to add the details of a new student who has recently joined the course.
- ***Deleting*** It means to remove (delete) a particular data item from the given collection of data items. For example, to delete the name of a student who has left the course.
- ***Sorting*** Data items can be arranged in some order like ascending order or descending order depending on the type of application. For example, arranging the names of students in a class in an alphabetical order, or calculating the top three winners by arranging the participants' scores in descending order and then extracting the top three.
- ***Merging*** Lists of two sorted data items can be combined to form a single list of sorted data items.

## ABSTRACT DATA TYPE:

An abstract data type, sometimes abbreviated ADT**,** is a logical description of how we view the data and the operations that are allowed without regard to how they will be implemented. This means that we are concerned only with what data is representing and not with how it will eventually be constructed. By providing this level of abstraction, we are creating an encapsulation around the data. The idea is that by encapsulating the details of the implementation, we are hiding them from the user's view. This is called information hiding**.** The implementation of an abstract data type, often referred to as a data structure**,** will require that we provide a physical view of the data using some collection of programming constructs and primitive data types

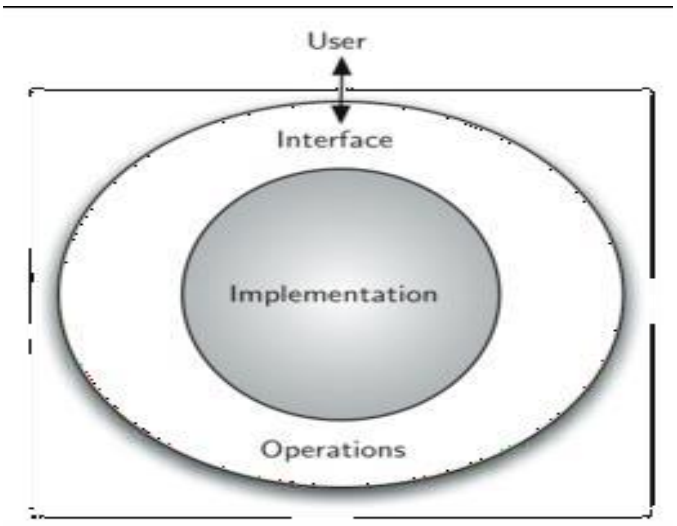
- An *abstract data type* (ADT) is a data structure, focusing on what it does and ignoring how it doesits job. (or) Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of value and a set of operations.




- Ex: stacks ADT and queues ADT. the user is concerned only with the type of data and the operations that can be performed on it.
- We can implement both these ADTs using an array or a linked list.

***Advantage of using ADTs***

- Modification of a program is simple, For example, if you want to add a new field to a student's record to keep track of more information about each student, then it will be better to replace an array with a linked structure to improve the program's efficiency.
- In such a scenario, rewriting every procedure that uses the changed structure is not desirable. Therefore, a better alternative is to *separate* the use of a data structure from the details of its implementation.



## PRELIMINARIES OF ALGORITHM:

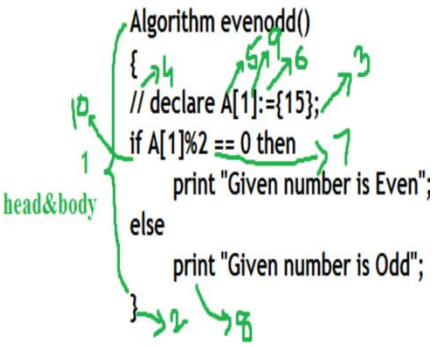
- Algorithm is step by step logical procedure for solving a problem.
- In Algorithm each step is called *Instruction.*
- An Algorithm is any well-defined computational procedure that take some values as inputs and produce some values as output.
- An Algorithm is a sequence of computational steps that transform input into output.
- An Algorithm has 5 basic properties:
  1. *Input:*An Algorithm has take '0' or more number of inputs that can be supplied as externally.
  2. *Output:* An Algorithm must produce at least one output.
  3. *Definiteness:* Each instruction in the algorithm must be clear.
  4. *Finiteness:* An algorithm must terminate after a finite number of steps.
  5. *Effectiveness:* Each operation should be effective. i.e the operations must be terminate after finite amount of time.

***Structure of an Algorithm:***

1. Algorithm is a procedure consisting of heading and body. In body part we are writing statements and in the head part we are writing the following.

   Syntax: Algorithm name_of_Algo (param1,param2, …);
2. The beginning and ending of block should be indicated by '{' and '}' or 'start' and 'end' respectively.
3. Every statement in the algorithm should be end with semicolon (;).

4. Single line




comments are written using '//' as beginning of comments.

5.The identifier should begin with character and it may be combination of alpha numeric.

5. Assignment operator (:=) we can use as follows

   Variable := expression (or) value;

6. There are other type of operators such as Boolean operators (TRUE/FALSE), logical operators (AND,OR,NOT) and relational operators (<,>,<=,>=,…..)
7. The input and output we can write it as read and print respectively.
8. The Array index are stored with in [ ] brackets. The index of array starts from '0' to 'N-1'.

   Syntax: datatype Aray_name[size];

9. The conditional statements such as if-then (or) if-then-else are written as follows.

```
if(condition) then statements;
if(condition) then
    statements;
else
    statements;
```

## TIME AND SPACE COMPLEXITY:

- The efficiency of an algorithm can be computed by measuring the performance of an algorithm. We can measure the performance of an algorithm in Two(2) ways.
  1. Time Complexity
  2. Space Complexity

### 1. Time Complexity:

- The time complexity of an algorithm is the amount of computing time required by an algorithm to run its completion.
- There are 2 types of computing time 1. Compile time 2. Run time
- The time complexity generally computed at run time (or) execution time.
- The time complexity can be calculated in terms of frequency count.
- Frequency count is a count denoting the number of times the statement should be executed.
- The time complexity can be calculated as

  Comments – 0
  Assignment / return statement – 1
  Conditional (or) Selection Constructs – 1

**Example 1: Sum of the elements in an Array**

| Statements | Step count/ Execution | Frequency | Total Steps |
|---|---|---|---|
| Algorithm Addition (A,n) | 0 | - | 0 |
| { | 0 | - | 0 |
| //A is an array of size 'n' | 0 | - | 0 |
| Sum :=0; | 1 | 1 | 1 |
| for i:=1 to n do | 1 | n+1 | n+1 |
| Sum:=Sum+A[i]; | 1 | n | n |
| return Sum; | 1 | 1 | 1 |
| } | 0 | - | 0 |
| **Total** | | **2n+3** | |

**Example 2: Subtraction of two matrices**

| Statements | Step count/ Execution | Frequency | Total Steps |
|---|---|---|---|
| Algorithm Subtract (A,B,C,m,n) | 0 | - | 0 |
| { | 0 | - | 0 |
| for i:=1 to m do | 1 | m+1 | m+1 |
| for j:=1 to n do | 1 | m(n+1) | mn+m |
| C[i,j] := A[i,j] – B[i,j]; | 1 | mn | mn |
| } | 0 | - | 0 |
| **Total** | | **2mn+2m+1** | |

**2. Space Complexity:**

- Space Complexity can be defined as amount of memory (or) space required by an Algorithm to run.
- To compute the space complexity we use 2 factors i. Constant ii. Instance characteristics.
- The space requirement S(p) can be given as $S(p) = C+S_p$

  Where C- Constant, it denotes the space taken for input and output.

  $S_p$ – Amount of space taken by an instruction, variable and identifiers.

**Example 1: Sum of three numbers**

```
Algorithm Add(a,b,c)
{
//a,b,c are float type variables
return a+b+c;
}
```

- The space required for this algorithm is: Assume a,b,c are occupies 1 word size each, total size comes to be **3.**

**Example 2: Sum of Array values**

```
Algorithm Addition (A,n)
{
//A is an array of size 'n'
Sum :=0;
for i:=1 to n do
      Sum:=Sum+A[i];
return Sum;
}
```

- The space required for this algorithm is:

  One word space for each variable then i,sum,n → 3

  For Array A[ ] we require the size → n

  Total space complexity for this algorithm is **S(p) ≥ (n+3)**

**What to Analyze in an algorithm:**

An Algorithm can require different times to solve different problems of same size

1. Worst case: Maximum amount of time that an algorithm require to solve a problem of size 'n'. Normally we can take upper bound as complexity. We try to find worst case behavior.
2. Best case: Minimum amount of time that an algorithm require to solve a problem of size 'n'. Normally it is not much useful.
3. Average case: the average amount of time that an algorithm require to solve a problem of size 'n'. Some times it is difficult to find. Because we have to check all possible data organizations

**SEARCHING:**

- Searching means to find whether a particular value is present in an array or not.
- If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array.
- However, if the value is not present in the array, the searching process displays an appropriate message and in this case searching is said to be unsuccessful.
- Searching techniques are ***linear search, binary search*** and ***Fibonacci Search***

## LINEAR SEARCH:

- Linear search is a technique which traverse the array sequentially to locate given item or search element.
- In Linear search, we access each element of an array one by one sequentially and see weather it is desired element or not. We traverse the entire list and match each element of the list with the item whose location is to be found. If the match found then location of the item is returned otherwise the algorithm return NULL.
- A search is successful then it will return the location of desired element
- If A search will unsuccessful if all the elements are accessed and desired element not found.
- Linear search is mostly used to search an unordered list in which the items are not sorted.

Linear search is implemented using following steps...

**Step 1 -** Read the search element from the user.

**Step 2 -** Compare the search element with the first element in the list.

**Step 3 -** If both are matched, then display "Given element is found!!!" and terminate the function

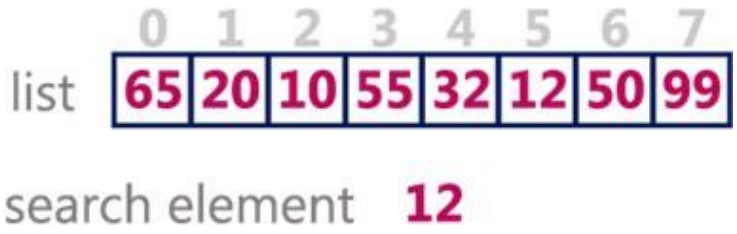
**Step 4 -** If both are not matched, then compare search element with the next element in the list.

**Step 5 -** Repeat steps 3 and 4 until search element is compared with last element in the list.

**Step 6 -** If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

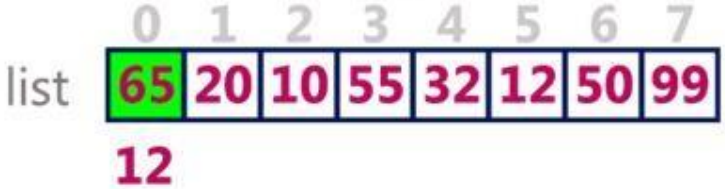
**Example:**

Consider the following list of elements and the element to be searched...




**Step 1:**

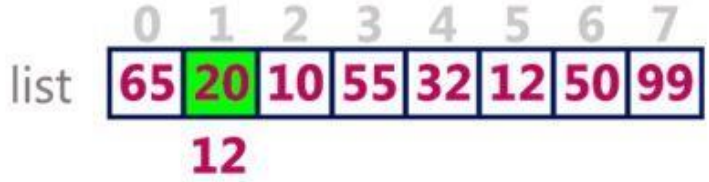
search element (12) is compared with first element (65)




Both are not matching. So move to next element

**Step 2:**

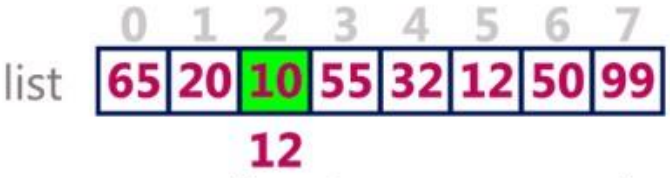
search element (12) is compared with next element (20)




Both are not matching. So move to next element

**Step 3:**

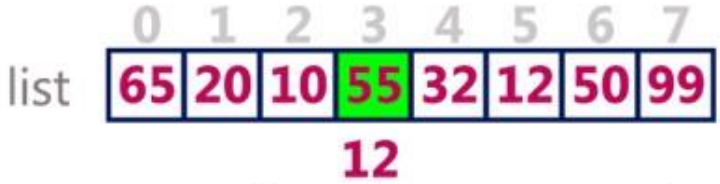
search element (12) is compared with next element (10)




Both are not matching. So move to next element

**Step 4:**

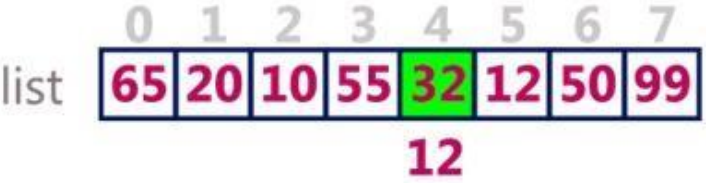
search element (12) is compared with next element (55)




Both are not matching. So move to next element

**Step 5:**

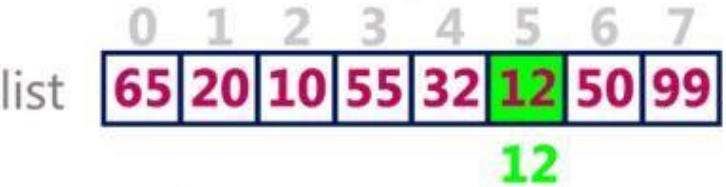
search element (12) is compared with next element (32)




Both are not matching. So move to next element

**Step 6:**

search element (12) is compared with next element (12)




Both are matching. So we stop comparing and display element found at index 5.

## BINARY SEARCH:

- Binary search is the search technique which works efficiently on the **sorted lists**. Hence, in order to search an element into some list by using binary search technique, we must ensure that the list is sorted.
- Binary search follows divide and conquer approach in which, the list is divided into two halves and the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves depending upon the result produced through the match.

Algorithm:

**Step 1 -** Read the search element from the user.

**Step 2 -** Find the middle element in the sorted list.

**Step 3 -** Compare the search element with the middle element in the sorted list.

**Step 4 -** If both are matched, then display "Given element is found!!!" and terminate the function.

**Step 5 -** If both are not matched, then check whether the search element is smaller or larger than the middle element.

**Step 6 -** If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.

**Step 7 -** If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.

**Step 8 -** Repeat the same process until we find the search element in the list or until sublist contains only one element.

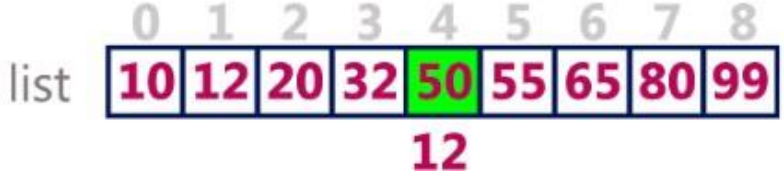
**Step 9 -** If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.

**Example:**


search element **12**

**Step 1:**

search element (12) is compared with middle element (50)



Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).


**Step 2:**

search element (12) is compared with middle element (12)

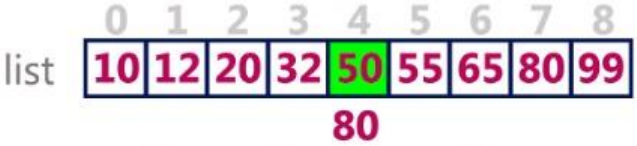

**Both are matching. So the result is "Element found at index 1"**

**Example 2:**

search element **80**

**Step 1:**

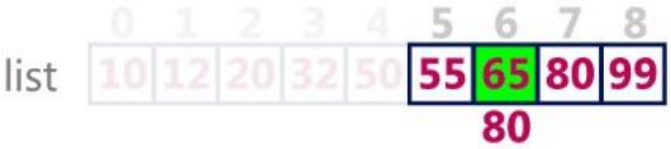
search element (80) is compared with middle element (50)



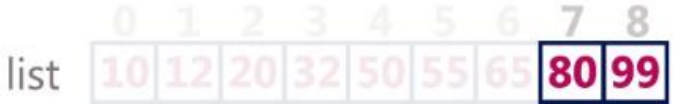

Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).


**Step 2:**

search element (80) is compared with middle element (65)

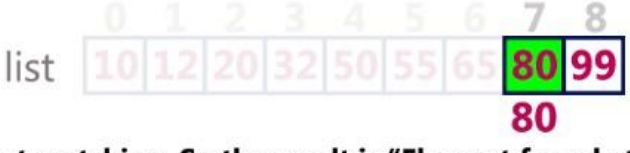



Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).




**Step 3:**

search element (80) is compared with middle element (80)




**Both are not matching. So the result is "Element found at index 7"**

## FIBONACCI SEARCH:

- **Fibonacci search** is an efficient search algorithm based on **divide and conquer** principle that can find an element in the given **sorted array** with the help of Fibonacci series in **O(log N)** time complexity. This is based on Fibonacci series which is an infinite sequence of numbers denoting a pattern which is captured by the following equation:

  **F(n)=n** **if n<=1**

  **F(n)=F(n-1)+F(n-2)** **if n>1**

  - where F(i) is the ith number of the Fibonacci series where F(0) and F(1) are defined as 0 and 1 respectively.
- The first few Fibonacci numbers are: **0,1,1,2,3,5,8,13....**

  F(0) = 0

  F(1) = 1

  F(2) = F(1) + F(0) = 1 + 0 = 1

  F(3) = F(2) + F(1) = 1 + 1 = 2

  F(4) = F(3) + F(2) = 1 + 2 = 3 and so continues the series
- Other searches like binary search also work for the similar principle on splitting the search space to a smaller space but what makes Fibonacci search different is that it divides the array in **unequal parts** and operations involved in this search are **addition and subtraction** these arithmetic operations takes place simple and hence **reducing the work load of the computing machine**.

**Algorithm:**

- Let the length of given array be **n [0. n-1]** and the element to be searched be **x**.
- Then we use the following steps to find the element with minimum steps:

**1.** Find the **smallest Fibonacci number greater than or equal to n**. Let this number be **f(M)**

Let the two Fibonacci numbers preceding it be **f(M-1)** and **f(M-2)**.

F(M) = F(Size of array)

F(M-1) = F(M) - 1

F(M-2) = F(M-1) -1

i (index) = min (offset + F(M-2) , n-1) //Offset = -1

2. While the array has elements to be checked:

-> Compare x with the last element of the range covered by f(M-2)

-> If **x** matches, return index value

-> Else if **x is less** than the element, move the three Fibonacci variables two Fibonacci down, Indicating removal of approximately two-third of the unsearched array from rear end. Not Reset offset to index

-> Else x is greater than the element, move the three Fibonacci variables one Fibonacci down. Reset offset to index. Indicating removal of approximately one-third of the unsearched array from front end.

3. Since there might be a single element remaining for comparison, check if F(M-1) is '1'. If Yes, compare x with that remaining element. If match, return index value.

**Example:** The Elements in array & Search key is

Search_Key 85

| elements | 10 | 22 | 35 | 40 | 45 | 50 | 80 | 82 | 85 | 90 | 95 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Initially the Fibonacci series is …

| **0** | **1** | **1** | **2** | **3** | **5** | **8** | **13** | **21** | **34** |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| | | | | | F(m-2) | F(m-1) | F(m) | | |

To calculate index position **i = min(offset+F(m-2), n-1),** Initially offset value is -1.

| **F(m)** | **F(m-1)** | **F(m-2)** | **Offset** | **i(index)** | **a[i]** | **Consequence** |
|---|---|---|---|---|---|---|
| 13 | 8 | 5 | -1 | (-1+5,10) = 4 | 45 | 1 steps down, Reset offset |
| 8 | 5 | 3 | 4 | (4+3, 10)=7 | 82 | 1 steps down, Reset offset |
| 5 | 3 | 2 | 7 | (7+2, 10) =9 | 90 | 2 steps down |
| 2 | 1 | 1 | 7 | (7+1, 10) = 8 | 85 | Return i |

Finally our desired element is **found at the location of 8.**

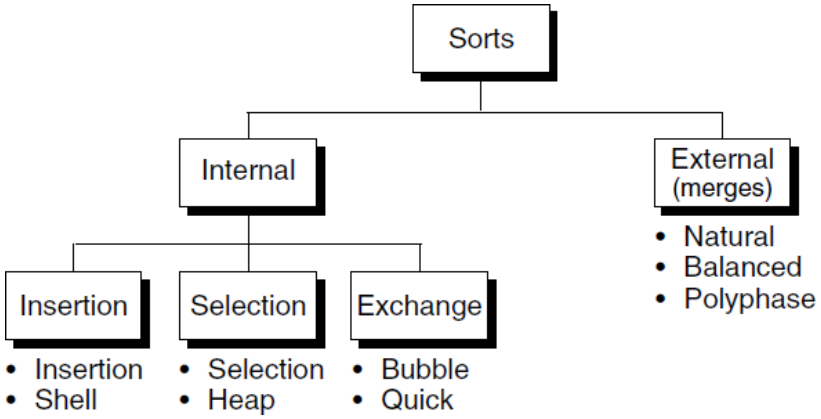
## SORTINGS:

- **Definition:** Sorting is a technique to rearrange the list of records(elements) either in ascending or descending order, Sorting is performed according to some key value of each record.

**Categories of Sorting:**

The sorting can be divided into two categories. These are:

- Internal Sorting
- External Sorting

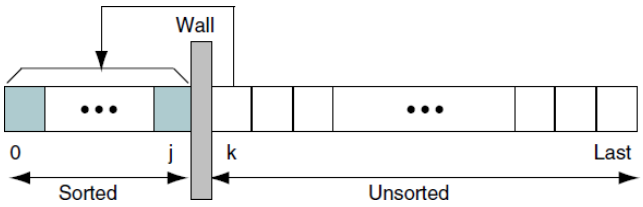


- **Internal Sorting:** When all the data that is to be sorted can be accommodated at a time in the main memory (Usually RAM). Internal sortings has five different classifications: insertion, selection, exchanging, merging, and distribution sort
- **External Sorting:** When all the data that is to be sorted can't be accommodated in the memory (Usually RAM) at the same time and some have to be kept in auxiliary memory such as hard disk, floppy disk, magnetic tapes etc.

  **Ex:** Natural, Balanced, and Polyphase.

## INSERTION SORT:

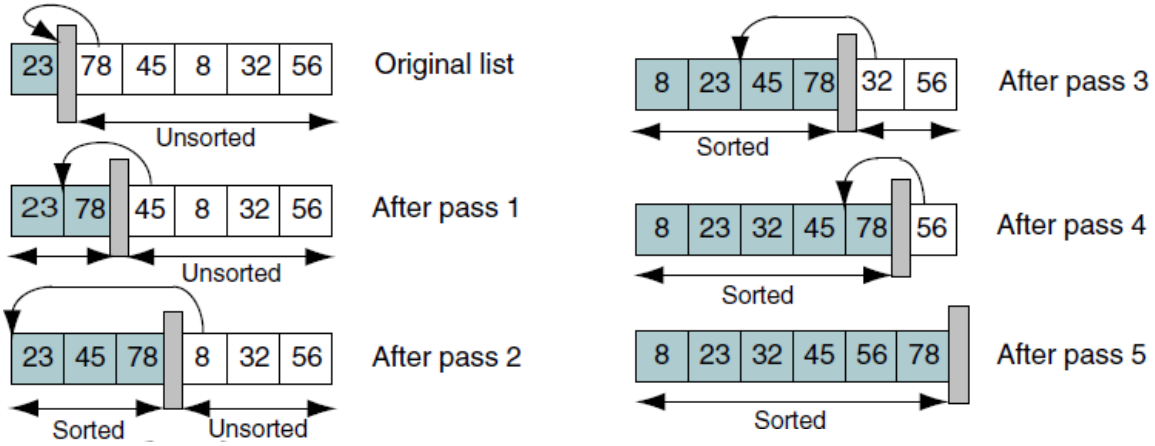
- In Insertion sort the list can be divided into two parts, one is sorted list and other is unsorted list. In each pass the first element of unsorted list is transfers to sorted list by inserting it in appropriate position or proper place.
- The similarity can be understood from the style we arrange a deck of cards. This sort works on the principle of inserting an element at a particular position, hence the name Insertion Sort.



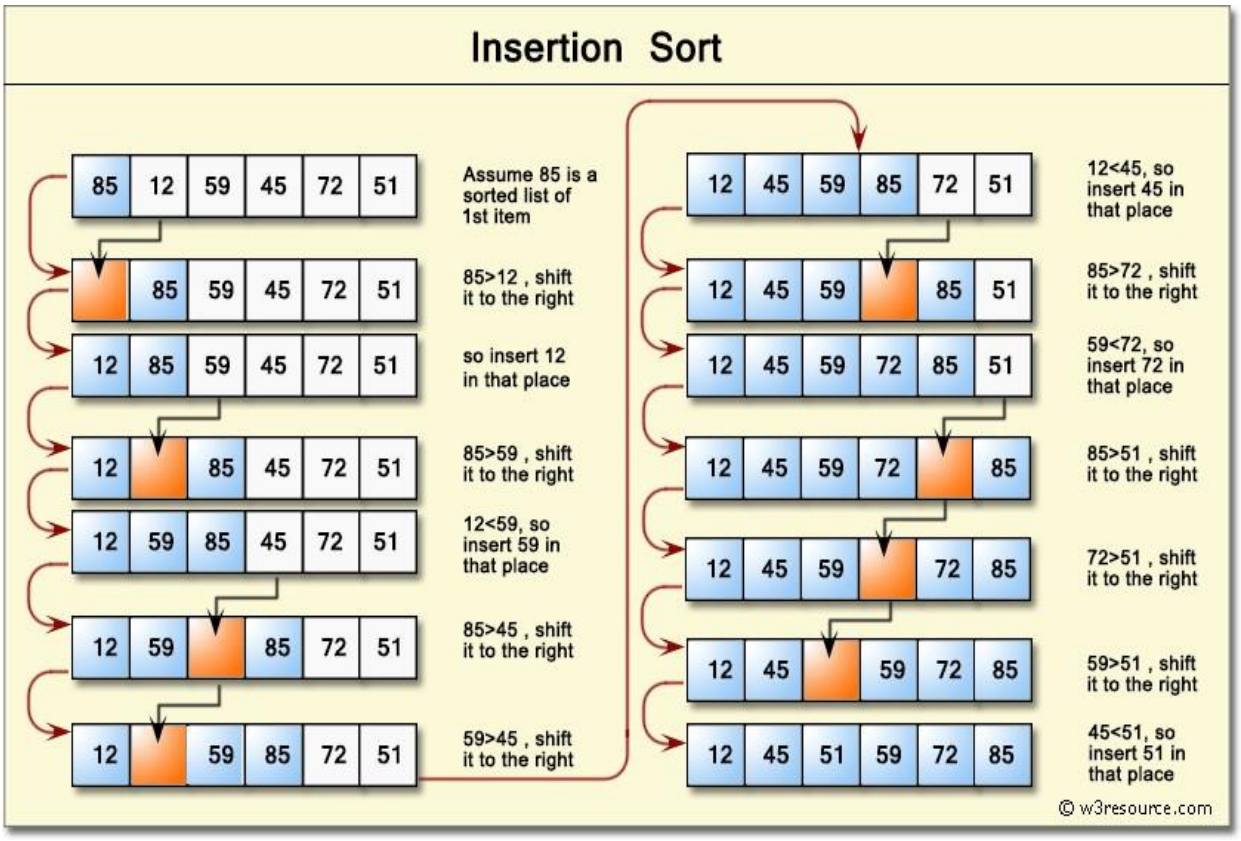

Following are the steps involved in insertion sort:

1. We start by taking the second element of the given array, i.e. element at index 1, the key. The key element here is the new card that we need to add to our existing sorted set of cards
2. We compare the key element with the element(s) before it, in this case, element at index 0:
   - If the key element is less than the first element, we insert the key element before the first element.
   - If the key element is greater than the first element, then we insert it after the first element.
3. Then, we make the third element of the array as key and will compare it with elements to it's left and insert it at the proper position.
4. And we go on repeating this, until the array is sorted.

**Example 1:**

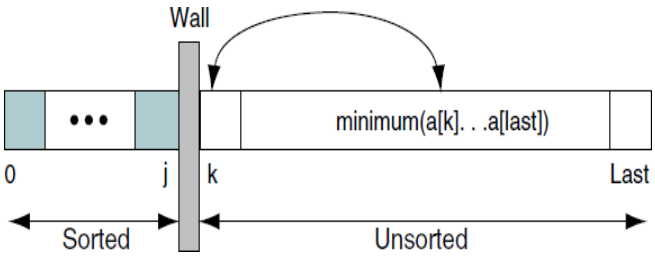


**Example 2:**




## SELECTION SORT:

- Given a list of data to be sorted, we simply select the smallest item and place it in a sorted list. These steps are then repeated until we have sorted all of the data.
- In first step, the smallest element is search in the list, once the smallest element is found, it is exchanged with the element in the first position.
- Now the list is divided into two parts. One is sorted list other is unsorted list. Find out the smallest element in the unsorted list and it is exchange with the starting position of unsorted list, after that it will added in to sorted list.




- This process is repeated until all the elements are sorted.

Ex: asked to sort a list on paper.

## Algorithm:

**SELECTION SORT(ARR, N)**

Step 1: Repeat Steps 2 and 3 for K = 1 to N-1

Step 2: CALL SMALLEST(ARR, K, N, Loc)

Step 3: SWAP A[K] with ARR[Loc]

Step 4: EXIT

**Algorithm for finding minimum element in the list.**

**SMALLEST (ARR, K, N, Loc)**

Step 1: [INITIALIZE] SET Min = ARR[K]

Step 2: [INITIALIZE] SET Loc = K

Step 3: Repeat for J = K+1 to N

IF Min > ARR[J]

SET Min = ARR[J]

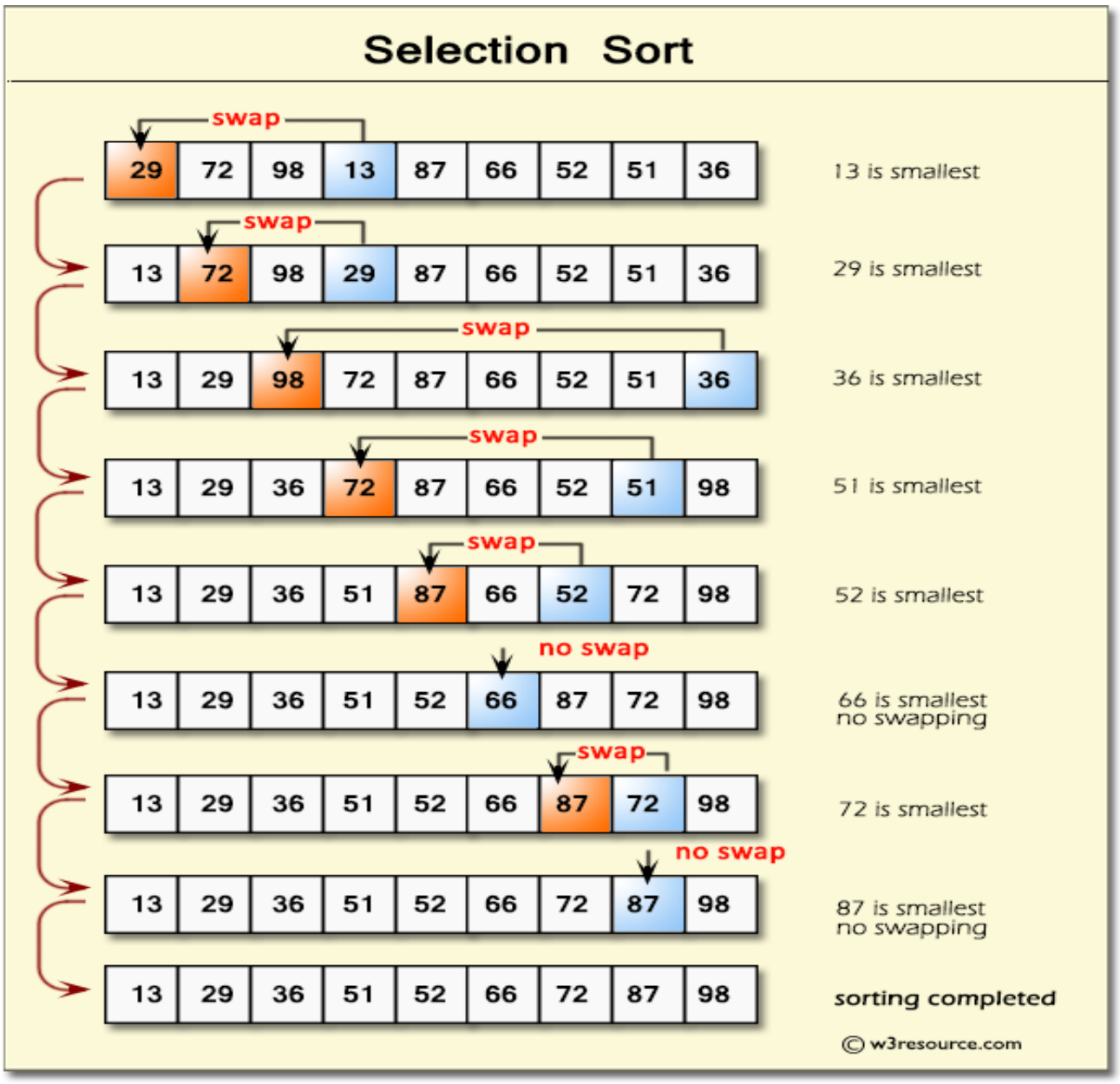
SET Loc = J

[END OF IF]

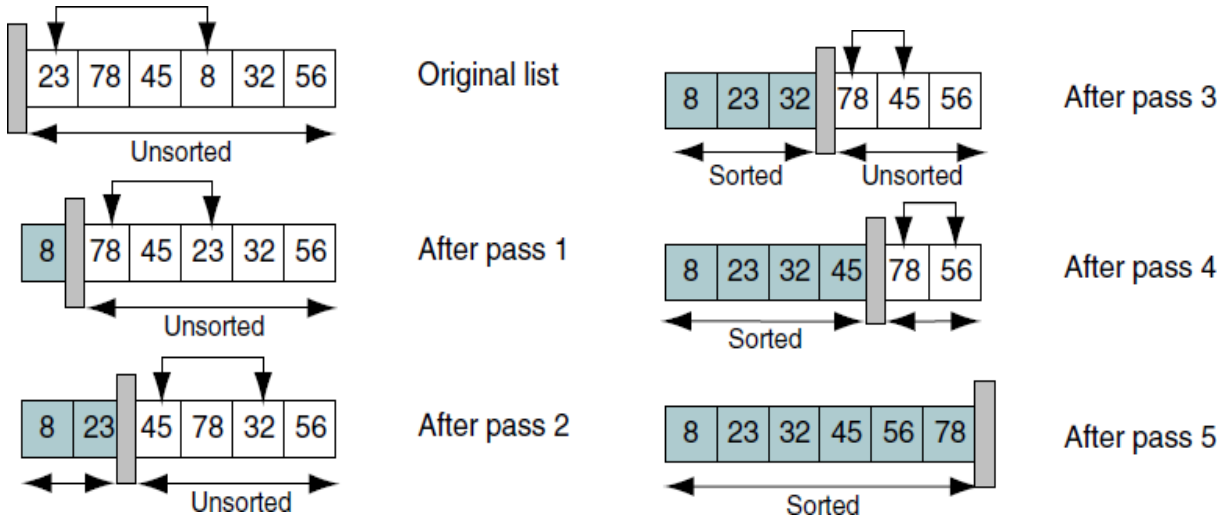
[END OF LOOP]

Step 4: RETURN Loc

**Example 1**:



**Example 2**: Consider the elements 23,78,45,88,32,56




**Time Complexity:**

Number of elements in an array is 'N'

Number of passes required to sort is 'N-1'

Number of comparisons in each pass is 1st pass N-1, 2nd Pass N-2 …

Time required for complete sorting is:

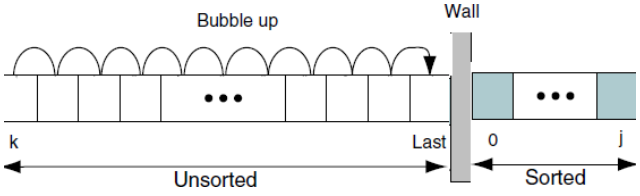
$T(n) <= (N-1)*(N-1)$

$T(n) <= (N-1)^2$

Finally, The time complexity is $O(n^2)$.

## BUBBLE SORT:

- Bubble Sort is also called as Exchange Sort
- In Bubble Sort, each element is compared with its adjacent element
  a) If he first element is larger than the second element then the position of the elements are interchanged.
  b) Otherwise, the position of the elements are not changed.
  c) The same procedure is repeated until no more elements are left for comparison.
- After the 1st pass the largest element is placed at $(N-1)^{th}$ location. Given a list of *n* elements, the bubble sort requires up to $n - 1$ passes to sort the data.




**Example 1:**

- We take an unsorted array for our example.

- Bubble sort starts with very first two elements, comparing them to check which one is greater.


- In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27. We find that 27 is smaller than 33 and these two values must be swapped.

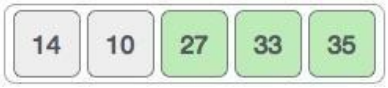

- Next we compare 33 and 35. We find that both are in already sorted positions.


- Then we move to the next two values, 35 and 10. We know then that 10 is smaller 35.

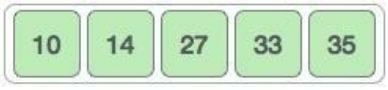

- We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −


- To be defined, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this

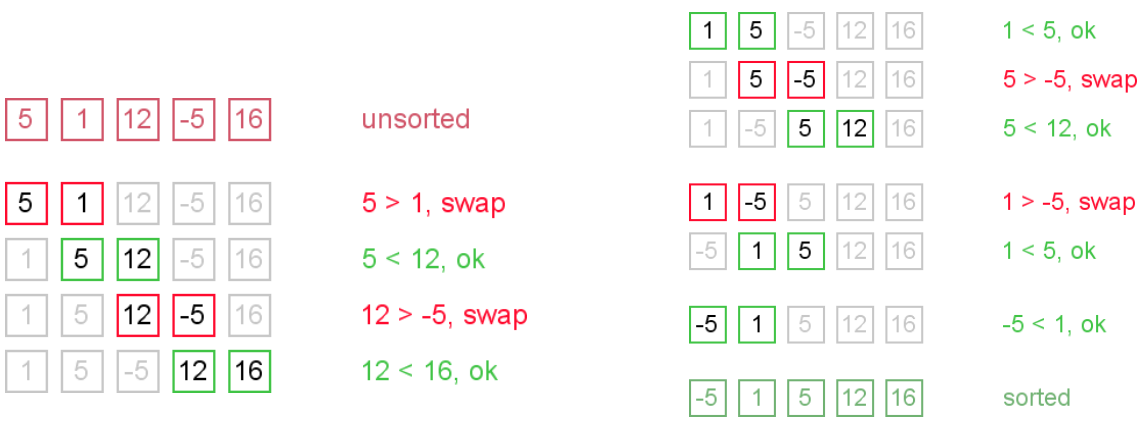

- Notice that after each iteration, at least one value moves at the end.




- And when there's no swap required, bubble sorts learns that an array is completely sorted.




**Example 2:**



**Algorithm**:

**BUBBLE SORT(ARR, N)**

Step 1: Read the array elements

Step 2: i:=0;

Step 3: Repeat step 4 and step 5 until i<n

Step 4: j:=0;

Step 5: Repeat step 6 until j<(n-1)-i

Step 6: if A[j] > A[j+1]

Swap(A[j],A[j+1])

End if

End loop 5

End loop 3

Step 7: EXIT

**Time Complexity:**

Number of elements in an array is 'N'

Number of passes required to sort is 'N-1'

Number of comparisons in each pass is 1st pass N-1, 2nd Pass N-2 …

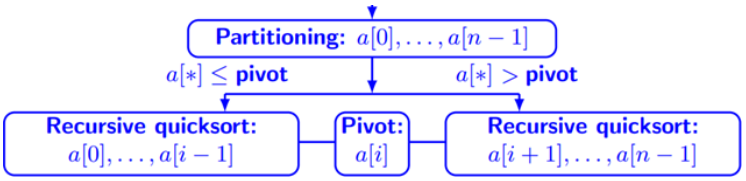
Time required for complete sorting is:

T(n) <= (N-1)*(N-1)

T(n) <= $(N-1)^2$

Finally, The time complexity is $O(n^2)$.

**QUICK SORT:**




- Quick sort follows **Divide and Conquer** algorithm. It is dividing array in to smaller parts based on partitioning and performing the sort operations on those divided smaller parts. Hence, it works well for large datasets.

So, here are the steps **how Quick sort** works in simple words.

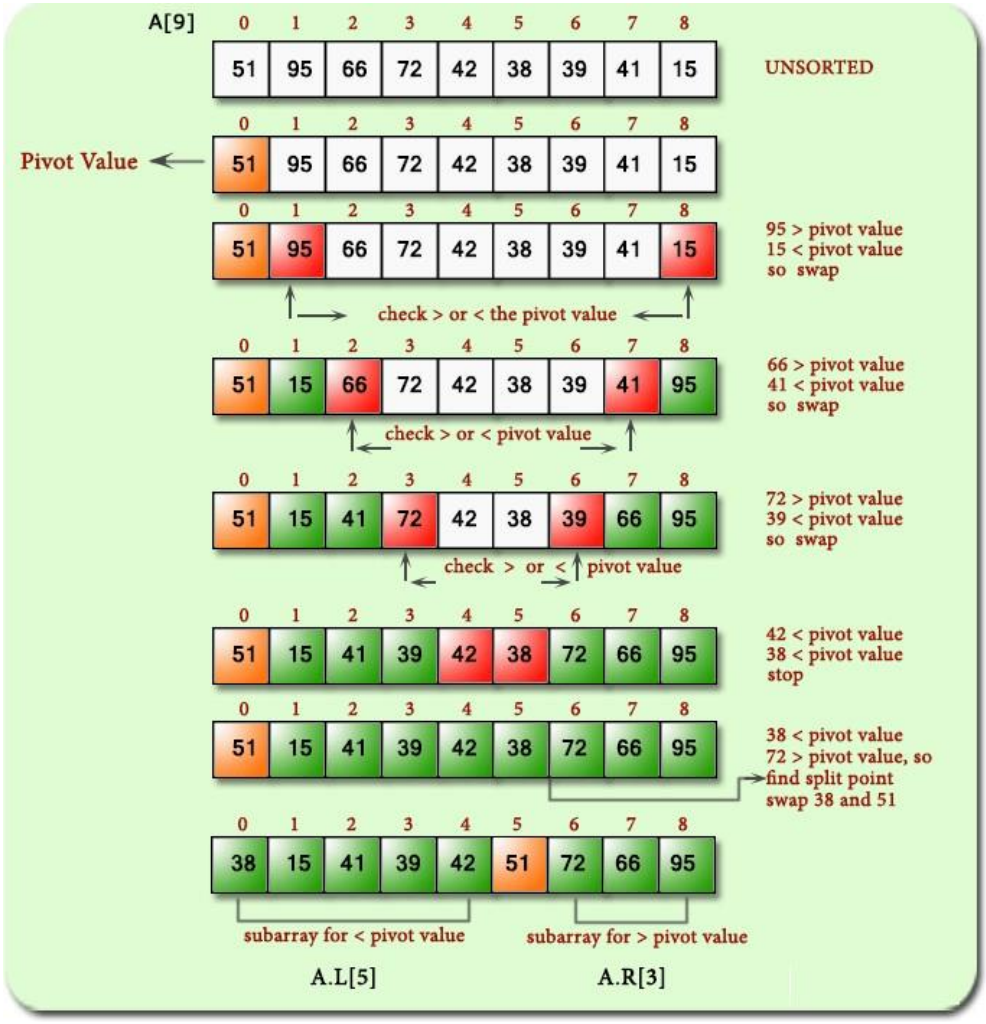
1. First select an element which is to be called as **pivot** element.
2. Next, compare all array elements with the selected pivot element and arrange them in such a way that, elements less than the pivot element are to its left and greater than pivot is to it's right.
3. Finally, perform the same operations on left and right side elements to the pivot element.

**How does Quick Sort Partitioning Work**

1. First find the **"pivot"** element in the array.
2. Start the left pointer at first element of the array.
3. Start the right pointer at last element of the array.

4. Compare the element pointing with left pointer and if it is less than the pivot element, then move the left pointer to the right (add 1 to the left index). Continue this until left side element is greater than or equal to the pivot element.
5. Compare the element pointing with right pointer and if it is greater than the pivot element, then move the right pointer to the left (subtract 1 to the right index). Continue this until right side element is less than or equal to the pivot element.
6. Check if left pointer is less than or equal to right pointer, then swap the elements in locations of these pointers.
7. Check if index of left pointer is greater than the index of the right pointer, then swap pivot element with right pointer.

**Example:**




**Algorithm:**

```
quickSort(array, lb, ub)
{
   if(lb< ub)
   {
      pivotIndex = partition(arr, lb, ub);
      quickSort(arr, lb, pIndex - 1);
      quickSort(arr, pivotIndex+1, ub);
   }
}
```

## RADIX SORT:

- Radix sort is a linear sorting algorithm for integers and uses the concept of sorting names in alphabetical order. When we have a list of sorted names, the *radix* is 26 (or 26 buckets) because there are 26 letters in the English alphabet. So radix sort is also known as bucket sort.
- Observe that words are first sorted according to the first letter of the name. That is, 26 classes are used to arrange the names, where the first class stores the names that begin with A, the second class contains the names with B, and so on.
- During the second pass, names are grouped according to the second letter. After the second pass, names are sorted on the first two letters. This process is continued till the $n^{th}$ pass, where n is the length of the name with maximum number of letters.
- When radix sort is used on integers, sorting is done on each of the digits in the number. The sorting procedure proceeds by sorting the least significant (LSD) to the most significant (MSD) digit. While sorting the numbers, we have **ten** buckets, each for one digit (0, 1, 2, …, 9) and the number of passes will depend on the **length** of the number having maximum number of digits.

**Example 1:** Sort the numbers given below using radix sort.

345, 654, 924, 123, 567, 472, 555, 808, 911

- In the first pass, the numbers are sorted according to the digit at ones place.

| Number | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 345 | | | | | | 345 | | | | |
| 654 | | | | | 654 | | | | | |
| 924 | | | | | 924 | | | | | |
| 123 | | | | 123 | | | | | | |
| 567 | | | | | | | | 567 | | |
| 472 | | | 472 | | | | | | | |
| 555 | | | | | | 555 | | | | |
| 808 | | | | | | | | | 808 | |
| 911 | | 911 | | | | | | | | |

- After this pass, the numbers are collected bucket by bucket. In the second pass, the numbers are sorted according to the digit at the tens place.

| Number | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 911 | | 911 | | | | | | | | |
| 472 | | | | | | | | 472 | | |
| 123 | | | 123 | | | | | | | |
| 654 | | | | | | 654 | | | | |
| 924 | | | 924 | | | | | | | |
| 345 | | | | | 345 | | | | | |
| 555 | | | | | | 555 | | | | |
| 567 | | | | | | | 567 | | | |
| 808 | 808 | | | | | | | | | |

- In the third pass, the numbers are sorted according to the digit at the hundreds place.

| Number | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 808 | | | | | | | | | 808 | |
| 911 | | | | | | | | | | 911 |
| 123 | | 123 | | | | | | | | |
| 924 | | | | | | | | | | 924 |
| 345 | | | | 345 | | | | | | |
| 654 | | | | | | | 654 | | | |
| 555 | | | | | | 555 | | | | |
| 567 | | | | | | 567 | | | | |
| 472 | | | | | 472 | | | | | |

- The numbers are collected bucket by bucket. After the third pass, the list can be given as final sorted list. 123, 345, 472, 555, 567, 654, 808, 911, 924.

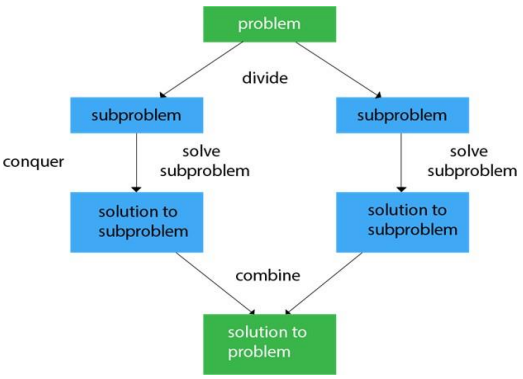
**Algorithm:**

1. Let **A** be a linear array of **n** elements **A[1], A[2], A[3] .....A[n]**. Digit is the total number of digit in the largest element in array **A**.
2. Input n number of elements in an array A.
3. Find the total number of digits in the largest element in the array.
4. Initialize i=1 and repeat the steps 4 and 5 until( i<=Digit).
5. Initialize the bucket j=0 and repeat the steps 5until (j<n).
6. Compare the $i^{th}$ position of each element of the array with bucket number and place it in the corresponding bucket.
7. Read the elements (S) of the bucket from $0^{th}$ bucket to $9^{th}$ bucket and from the first position to the higher one to generate new array A.
8. Display the sorted array A.
9. Exit.

**Divide and Conquer:**

- Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy.
- Divide and Conquer algorithm consists of a dispute using the following three steps.

1. **Divide** the original problem into a set of sub-problems.
2. **Conquer:** Solve every sub-problem individually, recursively.
3. **Combine:** Put together the solutions of the sub-problems to get the solution to the whole problem.




**MERGE SORT:**

Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several sub lists until each sub list consists of a single element and merging those sub lists in a manner that results into a sorted list.

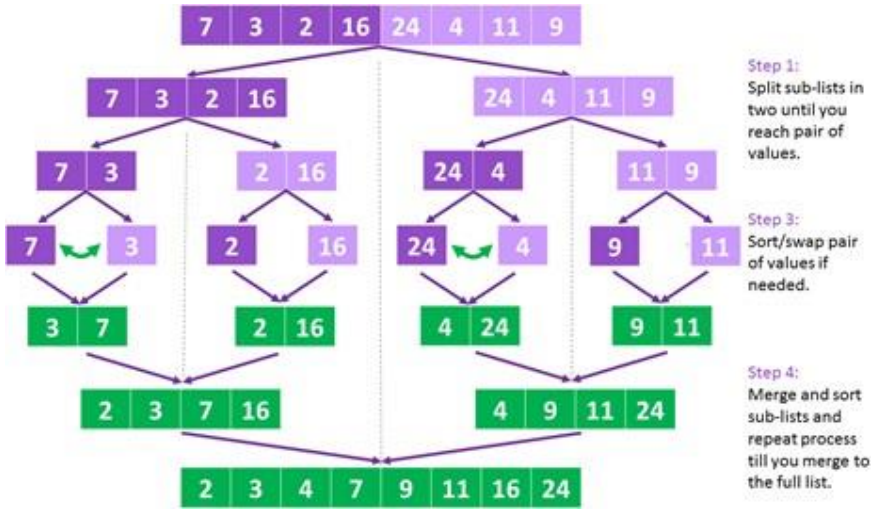
**Implementation Recursive Merge Sort:**

- The merge sort starts at the Top and proceeds downwards, "split the array into two, make a recursive call, and merge the results.", until one gets to the bottom of the array-tree.

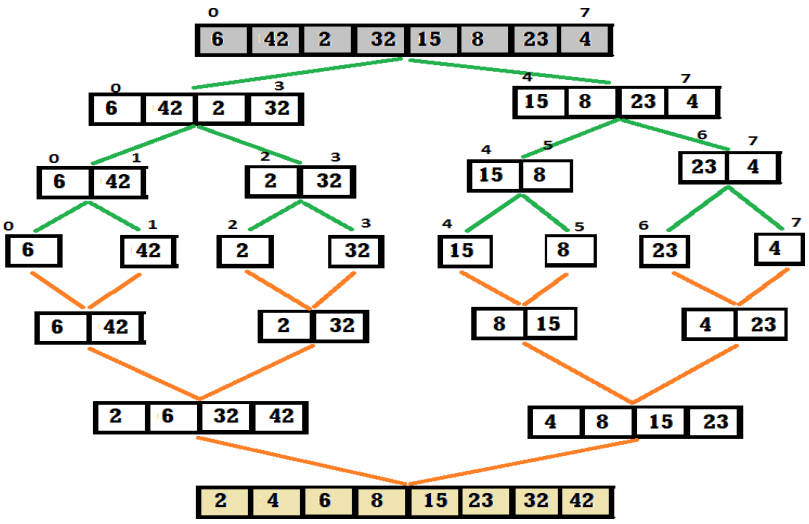
Example: Let us consider an example to understand the approach better.

1. Divide the unsorted list into n sub-lists based on mid value, each array consisting 1 element
2. Repeatedly merge sub-lists to produce newly sorted sub-lists until there is only 1 sub-list remaining. This will be the sorted list

**Recursive Mere Sort Example:**



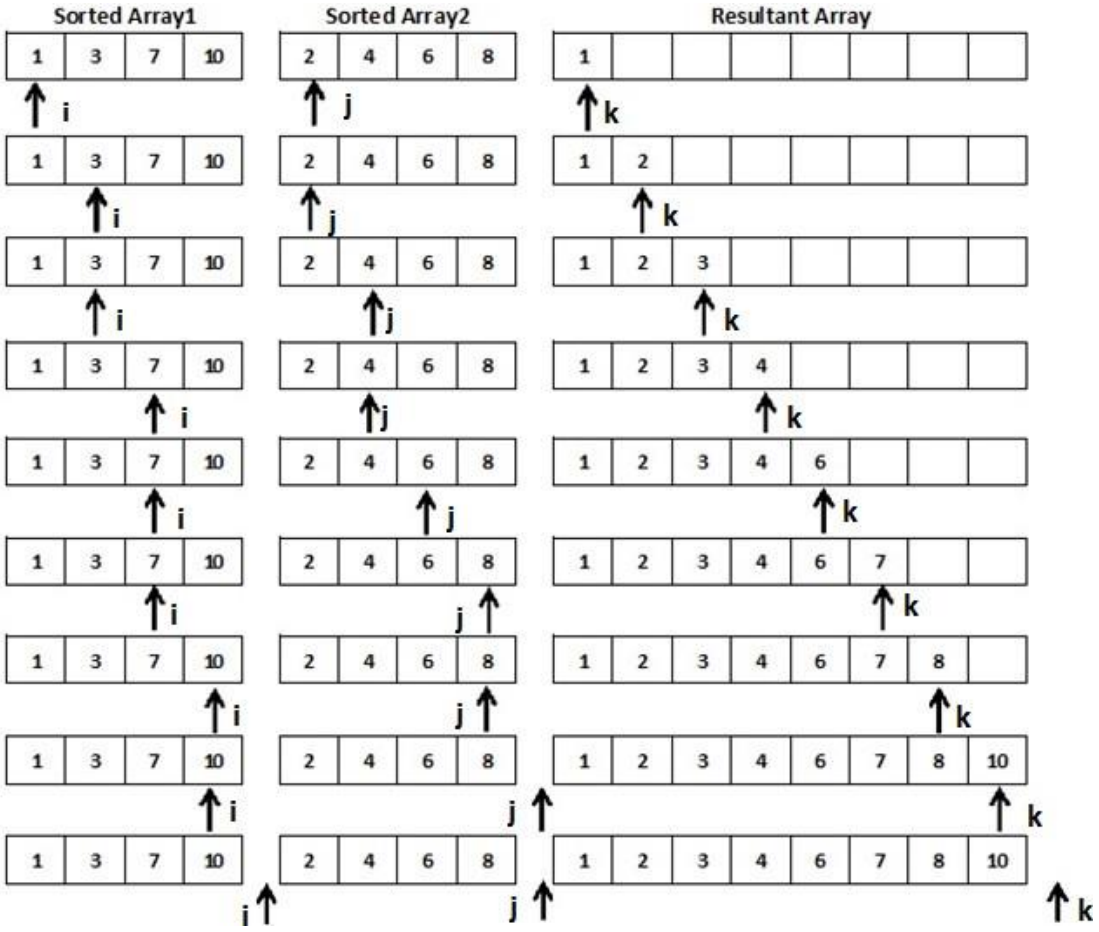

**Example 2:**




**MergeSort Algoritm:**

```
MergeSort(A, lb, ub )
{
   If  lb<ub
         {
            mid = floor(lb+ub)/2;
            mergeSort(A, lb, mid)
            mergeSort(A, mid+1, ub)
            merge(A, lb, ub , mid)
         }
}
```

**Two- Way Merge Sort:**




**Merge Algorithm:**

Step 1: set i,j,k=0

Step 2: if A[i]<B[j] then

copy A[i] to C[k] and increment i and k

else

copy B[j] to C[k] and increment j and k

Step 3: copy remaining elements of either A or B into Array C.

**<u>Time Complexities All the Searching & Sorting Techniques:</u>**

| Algorithm | Best Time Complexity | Average Time Complexity | Worst Time Complexity | Worst Space Complexity |
|---|---|---|---|---|
| Linear Search | O(1) | O(n) | O(n) | O(1) |
| Binary Search | O(1) | O(log n) | O(log n) | O(1) |
| Bubble Sort | O(n) | O(n^2) | O(n^2) | O(1) |
| Selection Sort | O(n^2) | O(n^2) | O(n^2) | O(1) |
| Insertion Sort | O(n) | O(n^2) | O(n^2) | O(1) |
| Merge Sort | O(nlogn) | O(nlogn) | O(nlogn) | O(n) |
| Quick Sort | O(nlogn) | O(nlogn) | O(n^2) | O(log n) |
| Radix Sort | O(nk) | O(nk) | O(nk) | O(n+k) |